Big Data Analytics Programming
Assignment 2: *Efficient Decision Tree Learning*

Pieter Robberechts
pieter.robberechts@kuleuven.be

Adem Kikaj
adem.kikaj@kuleuven.be

Laurens Devos
laurens.devos@kuleuven.be

Collaboration policy:
Projects are independent: no working together! You must come up with how to solve the problem independently. Do not discuss specifics of how you structure your solution, etc. You cannot share solution ideas, pseudocode, code, reports, etc. You cannot use code that is available online. You cannot look up answers to the problems online. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Decision Tree Learning

Decision trees are one of the oldest and most widely-used machine learning models. What makes decision trees special in the realm of ML models is their interpretability. The "knowledge" learned by a decision tree through training is directly formulated into a hierarchical structure. This structure holds and displays the knowledge in such a way that it can easily be understood, even by non-experts. Furthermore, they have low bias, as there is minimal implicitly defined structure in the model (as opposed to linear regression, for example, which makes the assumption of linear relationships). Unfortunately, this also means that decision trees are prone to high variance (there is no escaping the bias-variance trade-off) and will easily overfit on noisy data.

We will apply ensemble learning (more specifically, bagging) in an effort to minimize this weakness. This technique involves building a collection of models, trained on subsets of data, which are then combined to provide a robust model for classification or prediction. The basic idea is that each model provides a marginal amount of new information; when many of these models are combined, each providing some small incremental improvement to your prediction, you end up with a high-performing model.

In short, the goal of this assignment is to deliver an efficient C++ implementation of such an ensemble of decision tree learners. You will have to use CART as the underlying decision tree learning algorithm and the bagging technique to create the ensembles. Both algorithms are explained in more detail in the next sections.

1.1 The CART algorithm

There are many decision-tree algorithms, but we will implement the well known CART algorithm [Bre84]. Although the CART algorithm can be used for both classification and regression, we will only address the classification setting. Moreover, we will only consider discrete (both ordinal and categorical) features.

The CART algorithm acts by recursively partitioning the dataset into smaller, more homogeneous groups (i.e., that contain a larger proportion of one class in each partition). Starting from the root node, a binary tree is grown by repeatedly applying the following steps on each node:

1. Find each feature's best split.
   - Each split value v has a count matrix associated with it: the class counts in each of the partitions $x < v$ and $x \geq v$ (ordinal features) or $x \neq v$ and $x = v$ (categorical features). For each v, scan the database to gather this count matrix and compute its information gain (see below).
   - Determine the optimal splitting value (the one that maximizes the information gain).
2. Among the best splits of each feature, find the node's best split.
3. Split the node using the best split found in step 2. If $x < v$ (ordinal feature) or $x \neq v$ (categorical feature) the instance goes to the left node, otherwise to the right node. Next, recurse down the left and right sides.

To determine the optimal split points, CART uses the Gini index. If a data set S contains examples from c classes, the Gini index is defined as:

$$\text{Gini}(S) = 1 - \sum_{i=1}^{c} p_i^2, \tag{1}$$

where p_i is the relative frequency of class i in S. After splitting S into two subsets S_1 and S_2, the Information Gain (IG) of the split data is defined as

$$\text{IG}(S) = \text{Gini}(S) - \frac{|S_1|}{|S|}\text{Gini}(S_1) - \frac{|S_2|}{|S|}\text{Gini}(S_2). \tag{2}$$

The (feature, feature value) pair providing the largest $\text{IG}(S)$ is chosen to split the node.

The complexity of a naive implementation of this algorithm to construct a single level of the tree is $\mathcal{O}(F.E^2)$ – where F are the number of features and E is the number of examples. This is because we loop over the features ($|F|$), and for each one we loop over all examples to gather the count matrix. This is repeated for each unique feature value ($\mathcal{O}(|E|^2)$).

Luckily there are several well-known ways to speed this up. For example, one could sort the ordinal features once. Next you can linearly scan these values, each time updating the count matrix, and compute the information gain at each possible split point. Additionally, you can easily parallelize the recursive tree construction steps.

1.2 Bagging

Bagging [Bre96], short for bootstrap aggregation, is a general ensemble technique that uses bootstrapping to construct an ensemble. A bootstrap sample is a random sample of the data taken with replacement. This means that, after a data point is selected for the subset, it is still available for further selection. The

bootstrap sample is the same size as the original data set. As a result, some samples will be represented multiple times in the bootstrap sample while others will not be selected at all. The method is fairly simple in structure and consists of the steps in Algorithm 1. Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the bagged model's prediction.

Data: $m \leftarrow$ number of models
for $i \leftarrow 1$ **to** m **do**
    Generate a bootstrap sample of the original data;
    Train an unpruned tree model on this sample;
end

Algorithm 1: Bagging

1.3 Data

The data set used in this assignment is the well-known Covertype data set[1]. This data set records the types of forest covering parcels of land in Colorado, USA. Each example contains several features describing each parcel of land, like its elevation, slope, distance to water, shade, and soil type, along with the known forest type covering the land. The forest cover type is to be predicted from the rest of the features, of which there are 12 in total. It contains both categorical (string) and ordinal (integer) features. Your implementation does not have to support other types of features.

You can access the data directly on the departmental machines in the directory `/cw/bdap/assignment2/data/` and should not copy it to your machine. Note that the format of this dataset differs from the UCI version. We use the ARFF format[2] instead of CSV and the categorical features are not one-hot encoded. There are 463,859 examples in the train set and 117,153 in the test set. Code is provided that can read this dataset, but you can (and should) make some adaptations to allow more efficient decision tree learning.

Besides the Covertype data set, we included tree other small datasets in the `data` folder assignment's zip file: `iris` (only ordinal features), `tennis` (only categorical features) and `fruit` (both ordinal and categorical features). These can be used to test your implementation.

1.4 Implementation

All the class stubs and other code are available in `code/`. The folder `lib/` should contain your actual implementation, while the `test/` folder contains code to run this implementation on the dataset. Read these class stubs and understand them carefully. If your new to C++, this tutorial will help you to get started: `http://www.cs.ukzn.ac.za/~hughm/os/notes/c++ForJavaProgrammers.html`.

You should be careful to not only implement everything that is expected, but also, not to alter anything that is not supposed to change. Table 1 gives an overview of which classes can be modified. If some optimizations require small changes in the classes and types that should not be altered, you can still make minimal changes but you should add a small note to the class description describing your changes. Below we give some more details about the core of the implementation.

`DataReader` The `DataReader` class implements a parser for ARFF files. The method definitions of the public members of this class should not be altered, but you can change the implementation. Also the data types of the `Data` and `MetaData` aliases can be changed. Everything that is related to reading the data and

[1] https://archive.ics.uci.edu/ml/datasets/covertype
[2] https://www.cs.waikato.ac.nz/ml/weka/arff.html

Table 1: Summary of the classes and types that should be completed, can be modified or not altered.

Complete	**Can be modified**	**Do not alter**
DecisionTree	DataReader	Node
Bagging	Data	Leaf
Calculations	MetaData	Question
	Utils	TreeTest
		Dataset

converting it to an optimal format for decision tree learning can go in this class and is not included in the computation time.

`DecisionTree` A `DecisionTree` only stores a reference to it's root `Node`. You still have to implement the `buildTree(...)` method. Furthermore, the `DecisionTree` class provides functionality to print the learned tree.

`Node, Leaf and Question` These classes are used to represent the structure of a `DecisionTree`. A decision tree is a hierarchical structure in which each internal `Node` represents a "test" (`Question`) on an attribute, each internal node has two children which are the branches that represent the outcome of the test, and each leaf node represents a class label. You can add methods to these classes, but the existing code should not be modified.

`Calculations` This namespace should contain the actual computations that you have to perform to construct the decision tree. Everything in this namespace can be modified.

`TreeTest` This class provides the functionality to evaluate a decision tree on the test set and is the entry point for our automated tests. Do not modify the method definitions of this class. You can make small changes to the implementation if other parts of your implementation would break this class, but make sure the inputs and outputs don't change.

`Bagging` Implements bagging of decision trees. You still have to implement the `buildBag(...)` method. Furthermore, the `Bagging` class provides functionality to evaluate the ensemble on the test set.

1.5 Compiling and Running code

We use CMake to manage the build process. The `CMakeLists.txt` files that control the build process are provided. Make sure that you read and understand these files. You can adjust them if you wish (but make sure your code compiles on the departmental machines). First, standard build files should be created from these configuration files:

```
cd test/ && mkdir build && cd build,
cmake ..
```

Note that you have to run these commands only once. Next, you can use your platform's native build tools for the actual building:

```
make -j10
```

And train a single decision tree or bagging classifier with:
`./DecisionTreeTest`, and
`./BaggingTest`,

Finally, before you submit, make sure that you can compile your implementation as a library. Therefore, you should uncomment everything in `test/CMakeLists.txt` and run the following commands:

```
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX:PATH=<path> ..
make -j10
make install
cd ../test/build
cmake -DCMAKE_PREFIX_PATH:PATH=<path> ..
make -j10
./ImportTest
```

The `<path>` should be substituted by the location where you would like to install your library. The default location is `/usr/local/lib/`, but you don't have the permission to write there. You can use any path in your home dir or `/tmp`.

1.6 Grading the implementation [26pts]

Your implementation will be graded based on speed (12 pts), correctness (12 pts) and coding style (2 pts). Table 2 gives an overview of the scale that we will use for grading the computation time of your implementation. We will grade speed based on the average time required to train the first five decision trees in an ensemble (time required to load the dataset and evaluate the classifier not included). Obviously, you won't score a 12 on speed, if your implementation is far from correct.

Table 2: Scale used for grading tree learning speed.

Implementation[*]	**Absolute computation time**	**Score**
	< 1s	12+1 (Bonus)
SkLearn (4.76s)	< 5s	12
Weka (13.21s)	< 15s	11
	< 30s	10
	< 1m	9
	< 2m	8
	< 4m	7
	< 8m	6
Presort + linear scan (10m)	< 15m	5
	< 30m	3
	< 1h	1
Naive implementation (> 1 day)	> 1h	0

[*] Avg. time required to build one tree, averaged over the first five trees in a bagged decision tree classifier. Benchmarks run on `gent.cs.kotnet.kuleuven.be`.

The "Naive implementation" in Table 2 corresponds to a straight implementation of the algorithm discussed above. This implementation effectively splits the dataset for each possible (feature, feature value) pair and computes the information gain. As you can see, a decent computation speed can be achieved by presorting the feature values and determining the best feature value to split on with a single linear scan of the dataset.

2 Report [4 pts]

You should write a small report of at most one page and a half (including Tables and Figures). First, this report should discuss how you optimized the decision tree learner. Clearly explain your implementation decisions and how/to what extent they contribute to an improved computation time. Second, you should conduct an experiment to determine the effect of varying the number of trees in the ensemble on the accuracy. Design an experiment to test this question, and then provide a summary of the findings (e.g., increasing the number of ensemble causes..., because...).

In addition, the report can include a maximum of half an extra page describing any problems (that is, bugs) in your code and what may have caused them as well as any special points about your implementation.

References

[Bre84] Leo Breiman. *Classification and regression trees*. Routledge, 1984.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, Aug 1996.

3 Important remarks about grading and submitting the assignment

Deadline The assignment should be handed in on Toledo before **Friday the 2nd of April at noon**. There will be a 10% penalty per day, starting from the due date.

Deliverables You should upload an archive (`.zip` or `.tar.gz`) with the following structure and files:

```
assignment2/
├── report.pdf
└── code/
    ├── CMakeLists.txt
    └── lib/
        ├── CMakeLists.txt
        ├── include/
        │   └── *.hpp
        └── src/
            └── *.cpp
```

The C++ header and source files should contain complete source code for training a model on the full dataset. Also, make sure you submit the CMake configuration files, and – if needed – modify them such that your code compiles on the departmental machines. The report with the learning curve must be submitted in `.pdf`. Do not upload any data or output files. Also the `test` folder does not have to be submitted.

Evaluation Your solution will be evaluated based on the correctness of your results, the speed of your code, and the style of code.

Automated grading Automated tests are used for grading. Therefore you must follow these instructions:

- Use the exact filenames as specified in the assignment.
- Some parts of the provided interface/skeletons can't be modified. Make sure to respect this.
- Make sure your code compiles as a library and the `./TestImport` works.
- Your code must run on the departmental machines, this can be done over ssh and implies that your implementation cannot use more than 2GB of memory.
- Never use absolute paths.

If you fail to follow these instructions, you will lose points.

Running experiments This is a class about big data, so running experiments could take hours or even multiple days. You can do this easily on the departmental machines: it takes several minutes to the start the experiments and then you check back later to see the results. Failure to run and report results on the full data set will result in a substantial point reduction.